DATE 12 08 2021

Assignment #1

(1)

Asymptotic notations are methods / languages using which we can define the running time of algorithm based on input size.
To represent the upper & lower bounds, we need some kind of syntax, & this is represented in form of function $f(n)$.

- Logarithmic $\rightarrow \log_2 n$, Linear $\rightarrow n$
- Quadratic $\rightarrow n^2$, Polynomial $\rightarrow n^2$
- Exponential $\rightarrow a^n$

(2)

for(i=1 to n) { i = i*2; }
'i' is doubling ~~greater~~ everytime

for k^{th} step $\rightarrow 2^k = n$ & for (k+1) we are out of loop
taking log both side.

$\log_2 2^k = \log n$

$k = \log_2 n$; Time Complexity - $O(\log n)$
Ans

(3) $T(n) = 3T(n-1)$ if $n > 0$, otherwise 1.

~~$T(n) = 3(3T(n-1))$~~

~~$= 3$~~

$T(n) = aT(n-b) + f(n)$ [Master Theorem]

$a = 3$, $b = 1$

$\because f(n) = 0$, $k = 0$ ✓

$T(n) = O(n^k a^{\frac{n}{b}})$

$= T(n) = O(n^0 a^n)$

$\boxed{T(n) = O(3^n)}$

12/08

DATE

(4) $T(n) = 2T(n-1) - 1$

$T(n) = aT(n-b) + f(n)$

$a = 2$

$b = 1$

$T(n-1) = 2T(n-2) - 1$

$T(n) = 2(2T(n-2) - 1) - 1$

$= 4T(n-2) - 3$

Similarly

$T(n) = 8T(n-3) - 7$

$T(n) = 2^k T(n-k) - (2^k - 1)$

let $n - 1 = 1$

$T(n) = 2^k T(1) - 2^k + 1$

$= 2^k (T(1) - 1) + 1$

$\Rightarrow T(n) = O(2^k) = O(2^n)$

Ans

(5)

```
int i = 1, s = 1;
while (s <= n) {
    i++; s = s + i;
    printf("#");
}
```

we can see that 's' is increasing by (1)

$\therefore O(n)$

(6) void function (int n){
int i, count = 0;
for (i = 1; i*i <= n; i++)
count ++;

n = (5)

i = 1 1*1 <= n
i = 2 2*2 <= n
3 × out of loop

loop is working for n/2 time only.

∴ $O(n/2) \Rightarrow O(n)$ Ans.

(7) void function (int n) {
int i, j, k, count = 0;
for (i = n/2; i <= n; i++)
for (j = 1; j <= n; j = j * 2)
for (k = 1; k <= n; k = k * 2)
count ++

i loop = $O(n/2)$
j loop = $O(\log n)$
k loop = $O(\log n)$

∴ ~~Of~~ final = $O(n \cdot \log^2 n)$

Ans

(8) fun (int n) {
if (n == 1) return;
for (i = 1 to n)
for (j = 1 to n)
print (*);
}
}
func (n - 3);
}

(8) Ans
(2) (1)

Time complexity

$\boxed{T(n) = T(n^2) - 3}$

Ans

(9) void func(int n) {

for(i = 1 to n) $\rightarrow O(n)$

for(j = 1 to j <= n; j = j + i) $\rightarrow O(\log n)$

printf("*");

$\therefore O(n \log n)$ Ans.

(10)

$f(n) = n^k$ $\quad k >= 1$

$g(n) = a^n$ $\quad a > 1$

2^2

2^2

2^3

2^4

2^5

is $f(n) = O(g(n))$

$n^k = O(a^n)$

~~take~~ : $\log_a$

$k \log_a n = n \log a$

$= \frac{\log n}{\log a} = \frac{n}{k}$

$\log_a n = \frac{1}{k} n$

let $\boxed{1/k = c}$

$\therefore O(cn)$

is time complexity.

DATE

n=5

(11) void fun (int n){ int j=1, i=0;
while (i<n){ i = i+j , j++; } }.

→

i	j	n loop runned.
0	1	1.
1	2	2.
3	3	3.

we can observe that loop is running for $n/2 + 1$ times.

$\therefore O(n/2+1)$

$= O(n)$ Ans.

(12) int fin(int n){

Line 1 if (n==0 || n==1) return n;

Line 2 else return fib(n-1) + fib(n-2);

we know that line 1 takes O(1) time while line 2 takes $T(n-1) + T(n-2)$

$\therefore$ recursive eqn $= T(n-1) + T(n-2) + O(1)$

$\therefore T(n) = T(n-1) + T(n-2)$ — (1)

$T(n-1) = T(n-2) + T(n-3)$ — (2)

$T(n) = T(n-2) + T(n-2) + T(n-3)$ — (3)

$T(n-2) = T(n-3) + T(n-4)$ — (4)

$T(n) = 2T(n-3) + T(n-4)$

$T(n) = \cancel{(k-1)T(n-k) +}$

$2T(n-k) + T(n-(k+1))$

$= \cancel{2T(n-k) +}$

$O(n) = 2^n$

Space $= O(n)$

(13) (1) (n log n)

```
void f(int n){
    for(i=0 ; i<n ; i++)
        for(j=0; j<n ; j=j*2; )
            count++;
```

(2) n^3

```
for(i = 0, i<n; i++)
    for(j=0; j<n; j++)
        for( k=0 ; k<n; k++)
        {   print("GEO");
        }
```

(3) log(log n)

```
for( i=0 ; i<logn; i=i*2)
    print("Hi");
```

Scanned by CamScanner

(14) $T(n) = T(n/4) + T(n/2) + cn^2$

using Master's Theorem

we know that; $T(n/2) >= T(n/4)$

$\therefore$ $T(n) <= 2T(n/2) + cn^2$

Apply Master's theorem to RHS

$T(n) <= \Theta(n^2)$

$T(n) = O(n^2)$

Also

$T(n) >= cn^2$

$T(n) = \Theta(n^2)$

$T(n) = \Omega(n^2)$

Since

$T(n) = O(n^2)$

$T(n) = \Omega(n^2)$

$\therefore T(n) = \Theta(n^2)$ Ans

(15) ~~what~~

```
int fun(int n){
    for(i=1; i<=n; i++)
        for(j=1; j<n; j=j+i){ O(1);}
```

→ for the i loop = $O(n)$

for the j loop = $O(\log n)$

$\therefore O(n \cdot \log n)$ Ans

(16)
```
for(i=2; i<=n; i=pow(i,k))
{ // O(1)
}
```

n = 7

k = 2

i	
2	1
4	2
16	x

$2, 2^k, (2^k)^k, (2^{\cdot}$

$\therefore 2^{k \log_k \log(n)}$ or $2^{\log(n)}$

$= O(\log(\log n))$.

(17) quick sort

when quick sort repeatedly divides the array in to two parts of 99% and 1% it is the worst case.

:- Recurrence Relation :

Let current problem = n

• ↳ $(n-1)$ we have to call.

Quicksort Worst Case

$$T(n) = T(n-1) + T(1) + c \cdot n$$

$$\therefore$$

$$T(n) = T(n-1) + \Theta(n)$$

$$= \sum_{k=1}^{n} \Theta(k)$$

$$= \Theta\left(\sum_{1}^{n} k\right)$$

$$= \Theta(n^2) \text{ is the time complexity.}$$

Recursion Tree:

$n \longrightarrow n$

1 ↙ ↘ $n-1 \longrightarrow n$

1 ↙ ↘ $n-2 \longrightarrow n-1$

1 ↙ ↘ $n-3 \longrightarrow n-2$

1 ↙

2 3

1 ↙ ↘ 1 2

$O(n^2)$

If the partioning is maximally unbalanced at every recursive step of algorithm.

(18)

(a) 100% loglgn < logn < nlogn < root(n) < log n! < n < n^2 < 2^n < 2^{2n} < 4^n < n!

(b)

(c)

(19)

```
void linearSearch ( int arr[], int n, int x)
{
        if ( arr[n/2] >=x ) {
            for (i=0 to n/2+1)
                if (arr[i] == x)
                    cout << found;
                    break;
        }

        else { for (i= n/2+1 ; i<n ; i++)
                { if (arr[i] == x)
                    cout << found;
                    break;
                }
        }
}
```

(20) Iterative :

```
for( i=1 to n-1)
    key ← arr[i]
    j = i-1
    while ( j >= 0 && arr[j] > key)

        arr[j+1] = arr[j]
        j = j-1;
    }
    arr [j+1] ← key;
```

Recursive

```
void insertion(arr, n)
{ if (n <= 1) return;

insertion (arr, n-1);
last ← arr [n-1];
j ← n-2;
while ( j >= 0 && arr[j] > last)
{
arr [j+1] ← arr [j]
j --;
}
arr [j+1] = last;
```

It is known as Online Sort because it does not need to know anything about values, it will sort and the info is requested WHILE the algo is running.

It grabs new value at every iteration.

examples → selection & Insertion.

(21)

Bubble Sort
Best = $O(n)$
Worst = $O(n^2)$
Avg = $O(n^2)$

Insertion Sort
Best = $O(n)$
Worst = $O(n^2)$
Avg = $O(n^2)$

Selection Sort
Best = $O(n^2)$
Worst = $O(n^2)$
Avg = $O(n^2)$

Merge sort
Best = $O(n \log n)$
Worst = $O(n \log n)$
Avg = $O(n \log n)$

Quick Sort
Best = $O(n \log n)$
Worst = ~~$O(n \log n)$~~ // $O(n^2)$
Avg = $O(n \log n)$

(22)

In Place	Stable	Online
Bubble	Insertion	selection
Insertion	Merge	Insertion
Selection	Bubble	
Quick, Heap		Offline
Not Inplace	Not Stable	
	Quick	Bubble, Quick
Merge	Heap	Merge

(23) Iterative Binary Search

```
int binarySearch (int arr[], int l, int r, int x)
{
    while (l <= r) {
        int mid = (l+r)/2;
        if (arr[mid] == x)
            return mid;

        else if (arr[m] < x)
            l ← mid+1
        else
            r ← mid-1
    }
    return -1;
}
```

Recursive Binary Search

```
int binarySearch (int arr[], int l, int r, int x)
{
    while (l <= r) {
        int mid = (l+r)/2;

        if (arr[mid] == x) return mid;

        else if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);

        else
            return binarySearch(arr, mid+1, r, x);
    }
    return -1;
```

Time Complexity			Space Complexity	
	Binary	Linear	Binary	Linear
Iterative →	O(log n)	O(n)	O(1)	O(1)
Recursive →	O(1) O(log n)	O(n)	O(1) or O(log n)	O(1)

Vardhman (O(log n))

(24) int bs (int arr, int l, int r, int x)

```
{   while ( l <= r ) {
        int mid = (l + r)/2;                      -> T(1)
        if (arr[mid] == x) return mid;
        else if (arr[mid] < x)                    -> T(n/2)
            return bs(arr, mid+1, r, x);
        else
            return bs( arr, l, mid-1, x);         -> T(n/
    }
    return -1
}
```

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 1$$

$$\boxed{T(n) = T\left(\frac{n}{2}\right) + 1}$$

Ans

Scanned by CamScanner